Ex. 4, Fuzzy logic and Callbacks

Stefan Karlsson

February 25, 2014

1 Introduction

In this exercise, we will synthesize images, and draw shapes. There are countless built-in functions in matlab to draw things. For cirles and rectangles, you can perform the following(leave the figure open after execution):

```
hf = figure;
% draw a square centered at (-0.5, 0.5)
h1 = rectangle('Position' ,[-0.5,0.5, 1,1]);

% draw a circle centered at ( 0.5, 0.5)
h2 = rectangle('Position' ,[ 0.5,0.5, 1,1],...
               'Curvature',[1 1])

% draw a disk   centered at ( 0, -0.5)
h3 = rectangle('Position' ,[ 0,-0.5, 1,1],...
               'Curvature',[1 1],...
               'FaceColor',[0 0 0]);
```

Like most visualization functions in Matlab, `rectangle` returns a handle to the object it creates in the current axes. We can use the `get` and `set` functions as we have already learned. The following snippet smoothly transforms the disk to a filled square:

```
figure(hf);
for c = linspace(1,0,50)
  set(h3,'Curvature',[c c]);   %brings 'Curvature' down on each iteration
  pause(0.1);
end
```

We will not use the `rectangle` function more in this exercise, even though it is the easiest and fastest way to draw circles and squares. We are interested in learning about 2D functions and matrix manipulations in this exercise, and will take image synthesis as examples.

2 Binary(logical) 2D functions

We have seen and used Matlab boolean logic expressions, and we know such expressions are automatically done elementwise. Recall the following example of synthesizing images with hard boundaries:

```
[y,x] = ndgrid(t);

binIm1 = x>0;           %binary image, each element either 1 or 0
binIm2 = y>0;

subplot(1,2,1)
imagesc2(t,t,binIm1);
title('x>0')

subplot(1,2,2)
imagesc2(t,t,binIm2);
title('y>0')
```

We can form more complicated boolean expressions:

```
binIm1 = (x>0) & (y>0);       %x bigger than 0, AND, y bigger than 0
binIm2 = (x>0) | (y>0);       %x bigger than 0, OR , y bigger than 0
binIm3 =   (x+y)>3 ;          % x+y bigger than 3
binIm4 = ~((x+y)>3);          %(x+y bigger than 3)... NOT

subplot(2,2,1)
imagesc2(t,t,binIm1);
title('(x>0) & (y>0)');

subplot(2,2,2)
imagesc2(t,t,binIm2);
title('(x>0) | (y>0)');

subplot(2,2,3)
imagesc2(t,t,binIm3);
title('(x+y)>3');

subplot(2,2,4)
imagesc2(t,t,binIm4);
title('~((x+y)>3)');
```

If we wish to know how many elements are in the shape(those who are value 1), we can simply sum the matrix. If each matrix element has width=height=1, the number of elements is the same as the area of the shape:

```
sum(sum(binIm1))
```

We have seen the function `sum` before, when we used it on vectors. Here, we call it twice, which might seem confusing. As always, look into Matlab documentation when you are unsure about a function. There, we can read that `sum` works on one dimension at a time, and that is why we have to call it twice. A simple example works better than any detailed explanation:

```
disp('make a 3-by-4 matrix of ones:');
A  = ones(3,4)

disp('sum along the columns, get 1-by-4 result:');
A1 = sum(A)
disp('sum along the final dimension, get total sum:');
A2 = sum(A1)
```

```
disp('choose which dimension is summed first:');
sum(A,1)
sum(A,2)
```

Many functions in Matlab work in this row-wise fashion. It is often very convenient. An equivalent way to get the sum of an entire matrix is to write:

```
sum(A(:))
```

The reason this works, is because we flatten out the matrix first, which is done by:

```
A(:)
```

2.1 Some Geometric Shapes

We recall that we can let x and y vary in the interval $[-1, +1]$, using:

```
res   = 40;
t     = linspace(-1,1,res);
[y,x] = ndgrid(t);
```

This will make any resulting matrix be of size `res`, which is the resolution of the image synthesis we are working with.

To generate a filled square with one corner at the origin, of some width `w`:

```
w  = 0.5;
binIm1 = (x>=0) & (x<w) & (y>=0) & (y<w);

imagesc2(t,t,binIm1);
title('a square');
```

By the the same principle, we can draw a filled rectangle of some width `w` and height `h`, centered at a position `p`. This is done by calling the provided function `DrawRectangle1.m`.

```
DrawRectangle1(0.6,0.3);
```

Open up the function `DrawRectangle1` *in the editor and make sure you understand the code!*

We remind ourselves that a disk of radius r, centered at (p_x, p_y) is given by:

$$(x - p_x)^2 + (y - p_y)^2 < r^2$$

We can rewrite this as:

$$r - \sqrt{(x - p_x)^2 + (y - p_y)^2} > 0$$

This is implemented in the function `DrawDisk1.m`, make sure you understand it. Execute the following:

```
DrawDisk1(0.8,[0 0]);     %make a disk of radius 0.8, centered at origin
```

The above example generated a shape that is only vaguely disk-like. The edges are jagged, due to the disk being approximated with low resolution. With more elements in the matrix (higher resolution) we get a closer approximation of the disk. This can be done by:

```
DrawDisk1(0.8,[0 0],160);
```

In both our examples above (drawing a rectangle and a circle), there is clearly some geometric shape that we can never perfectly describe by samples. The same principle was true of our audio synthesis: our vectors were sampled sound, and not sound waves. In both our audio and our image examples, we get closer to describing our true objects if we use more elements[1].

The functions `DrawRectangle2.m` and `DrawDisk2.m` have the added functionality of returning the area of the approximated shape, and it also draws the true shapes[2] in red over the binary image that approximates it:

```
[~,EstimARect, TrueARect] = DrawRectangle2(0.7,0.85,[0 0])
figure;
[~,EstimACirc, TrueACirc] = DrawDisk2(0.8,[0 0])
```

After running above code, `EstimARect` will contain the area of the shape in the binary image, and `TrueARect` contain the true area of the shape being approximated. Notice how we put a `~` on the spot of the first output. This is just to tell Matlab that we wont be using that output.

Open `DrawRectangle2.m` *and* `DrawDisk2.m` *and make sure you understand how they work.*

The area of the approximated shapes and the true shapes are not identical, but we should expect the area to get closer if we increase the number of elements. Phrased differently: as the resolution increase, the estimated area and the true area should converge. Lets generate a graph to investigate this. This is done by the provided function `testSynthesis.m`, which you run without arguments:

```
testSynthesis
```

Open `testSynthesis.m` *and make sure you understand how it works.*

2.2 Fuzzy logic

When we deal with Boolean logic, the world is completely described as black or white, true or false. With fuzzy logic, we allow for all shades of gray, and generally speak of a truth-ness level.

For this to work, we need some fuzzy decision function. Such an example is found in the provided function `fuzzyFunc.m`. The function does the operation `x>0` by default(the two following graphs are identical):

[1] increase the sampling frequency in audio, the resolution for images

[2] So, of course the drawn 'true shapes' are approximations as well, as finely as any machine can do it

```
t  = linspace(-1,1,1000);

subplot(2,1,1);
plot(t, t>0);
ylim([-0.5, 1.5])
title('t>0')

subplot(2,1,2);
plot(t, fuzzyFunc(t));
ylim([-0.5, 1.5])
title('fuzzyFunc(t)')
```

Lets modify the parameter(second argument) of `fuzzyFunc`:

```
close all;
t   = linspace(-1,1,1000);
fuzz= linspace(0,0.5,5);

plot(t, fuzzyFunc(t,fuzz(1)),...
     t, fuzzyFunc(t,fuzz(2)),...
     t, fuzzyFunc(t,fuzz(3)),...
     t, fuzzyFunc(t,fuzz(4)),...
     t, fuzzyFunc(t,fuzz(5)));

ylim([-0.5, 1.5]);
title('t > 0, at different fuzziness level');
legend(num2str(fuzz'));
```

2.3 Anti-aliasing

The shapes we have considered so far in our image synthesis have had sharp defining boundaries(binary images, only 2 values allowed), which we noticed especially clearly with the circles(jagged edges). A standard technique in computer graphics is called anti-aliasing, and it deals specifically with how to remove such unwanted effects in images. If we execute the following line...

```
DrawDisk2(0.8);
```

...and investigate the output, we see the source of the problem. The algorithm only sets pixels to 1 that are inside the red shape, and to 0 if they are outside. We want to set pixels that the circle cut partially to a value between 0 and 1. We can use fuzzy logic to implement this anti-aliasing.

For any Boolean expression `a>0`, we can replace it with a fuzzy version as `fuzzyFunc(a,b)`, for some fuzzy level `b` (for `b=0` it yields identical logic). We have generated our disk image using Boolean logic but lets upgrade it to be fuzzy. A version is provided as `DrawDisk3.m`, lets compare the outputs:

```
subplot(1,2,1);
DrawDisk2(0.9);
subplot(1,2,2);
DrawDisk3(0.9);
```

Open `DrawDisk3.m` *and make sure you understand how it works.*

We can perform the same test of convergence with the anti-aliased synthesis. Modify `testSynthesis.m` to use `DrawDisk3.m`, and run it. You should see a smoother converging curve, and you should make sure you understand why that is.

3 The Global Workspace

By now you should have a very good understanding of what the *base workspace*[3] is: It is the set of variables you have available to you on the command line and scripts in Matlab. The base workspace is visible in the desktop window called 'workspace' in the main matlab desktop. If you dont have the window shown, you can enable it in the menus, or you can type `whos` in the command prompt at any time to display the workspace.

You also know by now that each function has its own *local function workspace* where variables are created specifically for that function when its called. Whenever a function is finished executing, its local workspace is emptied.

Apart from the local function workspace and the base workspace, there is a third dangerous ***global workspace***. In the global workspace variables can be accessed from anywhere, at any time. It is where inexperienced programmers go insane. The global workspace may seem like an attractive place to put all your variables at first, but it is actually a swamp with vicious brain-eating bugs. Use this workspace sparingly. With that said, sometimes(rarely) using global variables is the best way to solve a problem.

Just as we can access the global workspace from anywhere, we can kill all the contents in it from anywhere by:

```
clear global
```

To create a variable on the global workspace:

```
global g
```

This will generate a variable `g` visible in your workspace. It should be an empty variable (its value: `[]`).

```
isempty(g)    %should give 1
g = 1;
isempty(g)    %gives 0
```

In a function file, you do not have access to any of the variables of the matlab desktop, but you do have access to the globals. For example, the following simple function (create it yourself in a new m-file):

```
function testGlobals1()
  global g;
  disp(['global variable g has value:' num2str(g)])
```

[3] also called desktop workspace

4 Callback Functions

Lets take our first steps towards making our programs interactive by handling keyboard input. Every figure that we create in Matlab has the potential to handle key presses. We just have to define exactly what is supposed to happen when a key is pressed. Here is a very simple example on how to setup a figure to handle a keyboard press (the message '*keyboard pressed*' will be echoed to the prompt every time you hit any key):

```
hFig = figure;
set(hFig, 'WindowKeyPressFcn','disp(''keyboard pressed'')');
```

Here we see once again how we set a property of an object (referenced by the handle `hFig`). In this case, the property 'WindowKeyPressFcn' is set to a string containing commands. These commands are executed as soon as a key is pressed while the figure is in focus. With this approach we can't handle different behavior due to specific keys. More useful is to specify a function to be called when a key is pressed. Such a function is called a *callback*, and a simple example is provided in `myKeypress1.m`:

```
function myKeypress1(hFig,evnt)
% hFig - handle to the figure object
% evnt - structure with info on what key was pressed
keyPressed = evnt.Key;             %the dot operator means that 'Key' is a member of 'evnt'

disp(['Keypressed: ' keyPressed ]);
```

We will **NOT** be interested in calling `myKeypress1` ourselves (not from the command prompt nor from any script or function). Instead, we want Matlab to call it for us (with the right arguments), at anytime we press a key. We can achieve this by the following lines:

```
hFig = figure;
set(hFig, 'WindowKeyPressFcn',@myKeypress1)
```

As long as the new figure is the current figure (the selected one) you will now be able to press any key and have it printed to the prompt. In the above snippet, we see the use of the `@` operator for the first time. This is how we make a so called *function handle*, which is how we make references to functions without calling them (more on this in later exercises).

As another example of a callback function for keyboard input, consider `myKeypress2.m`. You will be able to move the figure left and right by using the arrow keys if you set it up with a figure.

Set up `myKeypress2` ***as a keyboard callback in the same way as above examples, then open it and understand the code***.

A program can make use of a lot of keys, and the `if-elseif` construction in `myKeypress2` can be tedious. A more suitable way to control program flow

in a callback is using the `switch-case` construction. This is implemented in the example `myKeypress3.m`, which allows you to move the figure around using ALL the arrow keys.

Set up `myKeypress3` ***as a keyboard callback in the same way as above examples, then open it and understand the code***.

Matlab will usually interrupt everything in order to let callbacks execute fast - whatever is in the callback will execute close to the event that triggered it(keyboard in our case). Often, a main program is running and should be affected by the callback. For example, say that we have an animation and we wish to modify the speed of it interactively, using the up/down arrow keys. The provided `moveInf.m` illustrates how this can be achieved, run it as:

```
moveInf
```

Open and understand the code in `moveInf.m`. It uses a callback `myKeypress4`(open this as well) that performs the updating of a global variable `speed`. This variable stores information on the speed of the animation and is used in the loop of `moveInf`.

Tasks

Solutions that do not fill the following requirements **EXACTLY, PRECISELY AND TO THE LETTER** will not be considered:

- Send your solutions by email to:

  stefan.karlsson@hh.se

  subject: Matlab, Exercise `X`, `YourNames`

- Send all the files that are requested, no more and no less, in one single zip file per exercise, with NO sub-folders in the zip file. All the files, for all the tasks should be bundled into one zip file.

- Put the Names of the authors, in remarks, at the top of every m-file.

- Send the solutions within 2 weeks of every exercise session. That is, you have a two week deadline to hand it in.

- You will get 2 chances to send it in to me correctly.

Task 1

function `e4_1(res)` generates an animation of a disk moving in the path shaped as the character ∞. Argument `res` is the resolution of the synthesis. Default value `res=40`. The disk should be drawn with anti-aliasing using fuzzy logic as outlined in section 2.3.

In addition to the disk, the shape of the circle that is approximated should appear exactly over the position where the disk is currently drawn(thus, both circle and disk are animated together). The circle should be drawn in red.

The animation should continue forever, only to end if the user kills the figure. The following keyboard interface should be available for the user:

- Up/Down keyboard arrows: the speed of the animation increase/decrease.
- Left/Right keyboard arrows: the fuzziness of the boundaries increase/decrease.
- 'q'/'a' keyboard arrows: the radius of the disk increase/decrease.

Hand in ***ONLY*** the file `e4_1.m`
see next page for hints

Hints, Task 1

You should hand in ONLY 1 file, so you can put the callback function at the end of the same file. A shell of a solution looks like this(all in one single file):

```
function e4_1(res)
if nargin<1
    res = 50;end

%TODO: define globals for keyboard input

hFig  = figure('WindowKeyPressFcn',@keypress);
t     = 0;

%TODO: initialize the disk, get a handle
hold on
%TODO: circle initialization, get a handle

title('up/down[speed], left/right[fuzziness], Q/A[radius]');
xlim([-1,1]);
ylim([-1,1]);

%%%%%%%%  main loop: %%%%%%%%%%
while ishandle(hFig)
%  TODO: update the graphics correctly using handles
  pause(0.05);
end

% keyboard callback:
function keypress(hFig,evnt)
keyPressed = evnt.Key;

switch keyPressed
  %%% TODO: enter cases for keyboard input, set globals accordingly
end
```

Task 2(optional for grade 4)

A flash light effect can be implemented, by multiplying an RGB image with the fuzzy disk image. The radius of the disk is the area lit by the flash light.

Make a function `e4_2()` that generates an animated flash light effect.

```
im=imread('ngc6543a.jpg');
```

The flashlight should move in the same way as in `e4_1`, and you can control the fuzziness of the boundaries, radius of the light and speed with the same keyboard interface.

In addition to above, the fuzziness of the boundaries should be different for different color channels:

- red channel : has fuzziness `fuz`
- green channel : has fuzziness `2*fuz`
- blue channel : has fuzziness `4*fuz`

where `fuz` should be set by the keyboard interface.

hand in only the function file `e4_2.m`.

Hints, Task 2

The resolution of the image will now be different on the x and y axis. The resolution should be set by the size of the image. Make variable `resX` and `resY` for this.